Fall 2022 CS120 Project 1. Acoustic Link

Due Date: Oct. 16, 2022

(10 points + 7 points)

Suggested workload: 4~6 FULL days

Please read the following instructions carefully:

- This project is to be completed by each group **individually**.
- Submit your code through Blackboard. The submission is performed by one of the group members.
- Each group needs to submit the code **once and only once**. Immediately after TAs' checking.

Overview

The goal of this project is to set up a communication link (not necessarily 100% reliable) between your devices through acoustic signals. A high-level architecture of the program looks like:

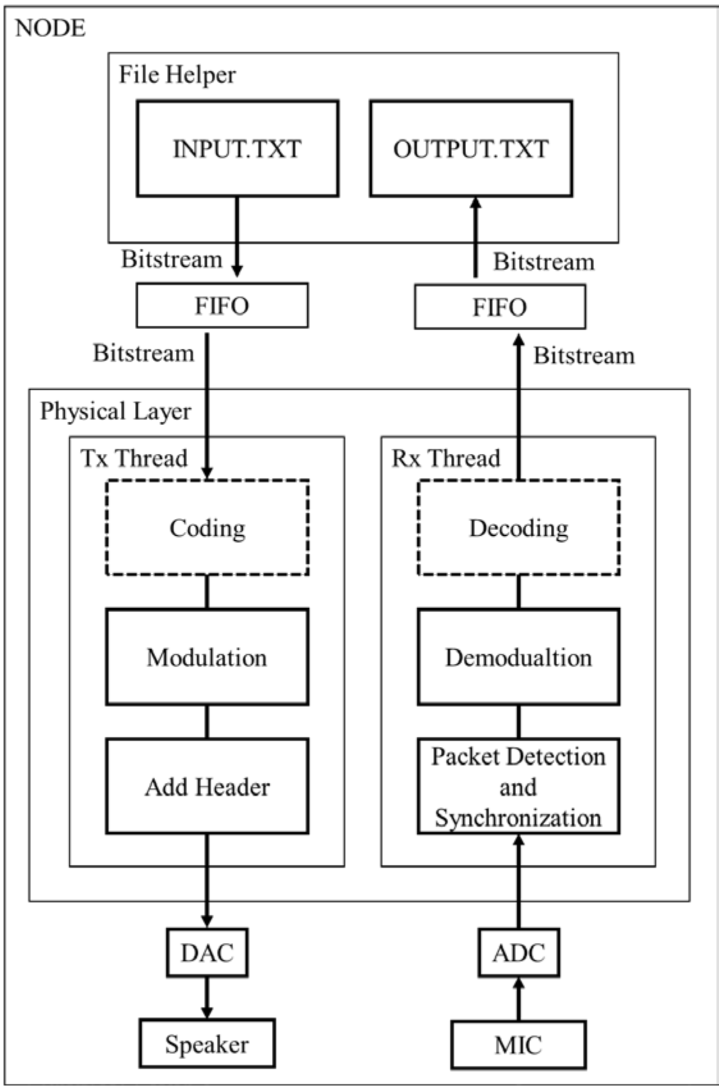

Figure 1 Project1 Overview

Part 1. (3 points) Understanding Your Tools

Different operating systems have different architectures for media I/O, but their usages are quite similar. In order to convey bits through acoustic signals, you have to correctly send and receive acoustic signals through the media interface.

Playing sound with a computer is about transmitting sound samples through the speaker. Recording is about sampling the sound signals through the microphone. You can start by referring to the Audio Stream Input/Output (ASIO) protocol [6]. ASIO is supported by most sound card drivers. Compared with the default/or common system audio interfaces, such as WindowsAudio or DirectSound, ASIO has a significant improvement in latency. Latency measures the elapse of time when a sample is "written" into the buffer of the sound interface and the time when the audio of that sample is actually played by the audio hardware. Latency matters a lot in our next project.

Latency is related to the buffer size of the sound interface. Choosing a smaller buffer size reduces latency, but might result in discontinuous sound in a loaded system. ASIO reduces the latency at its best while maintaining sound quality. ASIO has several driver implementations. ASIO4ALL [10] working in Windows is verified by the teaching team. For programming in JAVA, you can choose a JAVA wrapper [11] to access the ASIO driver. For programming in other languages, you can either use the ASIO native APIs directly or find other wrappers on your own. For Linux and MAC OS users, you can either try WineASIO or other low-latency audio interfaces, but there is no guarantee on that (the teaching team has not tested). Here is a utility to test the latency of the chosen sound interface [12]. The latency should be within 20 ms or better (15 ms) in order to finish Project 2.

In this part, your task is to use the chosen audio interface to correctly control your sound card to play digital samples through the speaker and record analog sound signals from the microphone. Playing and recording should be able to work simultaneously. You can use two separate threads for playing and recording.

Checkpoints:
The group provides one device: NODE1.
CK1(1.5 points). NODE1 is able to record the voice from TA (10 seconds) and then replay the recorded signals.
CK2(1.5 points). NODE1 is able to play a predefined sound (any) and record the playing sound at the same time. TA may say something during the recording. After 10 seconds, stop playing and recording. Then play the recorded sound for verification.

Tips:
a. Be careful about stereo settings. We only need one track
b. Be careful about the sampling rate of playback and recording, match them unless you know what you are doing.

Part 2. (2 points) Generating Correct Sound

Knowing how to play and record sound is the first step in generating a correct sound signal. The second step is more about calculation and matching the sampling rate. Suppose the sound you want to generate is f(t) and the DAC sample rate is *fs*. Filling digital samples f(0), f(1/*fs*), f(2/*fs*), …, f(n/*fs*) into the DAC buffer will generate correct sound signal f(t) for the speaker. In this procedure, f(t) is sampled into discrete samples at rate *fs*.

DAC's sample rates are fixed values such as 8kHz, 16kHz, 44.1kHz, 48kHz, etc. These values are internally determined by the hardware. Choose *fs* from the supported rates as your sample rate, since any sample rates other than the supported ones are to be converted by the operating system or the media player, which might lead to unexpected issues.

Checkpoints:
The group provides one device: NODE1.
CK1(2 points). NODE1 is able to play signal: f(t) = sin (2pi 1000 t) + sin (2pi 10000 t). TAs use spectrum analyzer to check the frequencies of the sound signal.

Tips:
a. You can record the sound into a file, and then use tools like Matlab/NumPy to load, plot and replay the sound in the file. It is a useful debugging approach that you will frequently use in this and future projects.
b. Window and Linux have very different architecture in handling sound [7]. You are recommended to use 48 kHz sample rate.
c. Useful phone Apps for debugging: ATG Lite (generating correct pure tone), SpectrumView (viewing sound wave and spectrum)

Part 3. (5 points) Transmitting Your First Bit

Modulation. You can choose one of the modulation methods from lecture slides or other resources (see wiki [9]). There is no best choice for all situations, but there exist good ones for certain situations. For our project, FSK is easy to implement and PSK can reach high throughput. ASK is not recommended. This project description uses PSK as an example.

In PSK, suppose phase 0 is used to represent “0” and phase pi is used to represent “1”. It is easy to imagine that the wave for conveying bits is the concatenation of several wave segments in Figure 2.

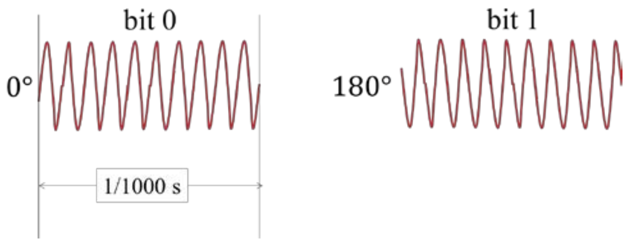

Figure 2 PSK Example

Further, suppose the carrier wave is 10 kHz and the bit rate is 1000 Hz, each wave segment in Figure 2 lasts for 1/1000 s, i.e., there are 10 cycles of the carrier wave in each segment. Similarly, if you want to reduce the bit rate to 500 Hz, each wave segment should last for 1/500 s, and there are 20 cycles in each segment. On the other hand, if the carrier wave is 20 kHz, the number of the cycles are doubled in the above description.

Now you know how to generate the modulated signals. The next step is to translate the signals into digital samples, so that DAC can play it. You have finished this step in Part2.

Adding a Header. A header of a frame is used to help the receiver to find out the accurate start of a frame, i.e. synchronize to the frame. Therefore, the header is normally a predefined special wave pattern that can help with synchronization. After adding the header, samples that you are going to fill into the DAC may have the structure in Figure 3:

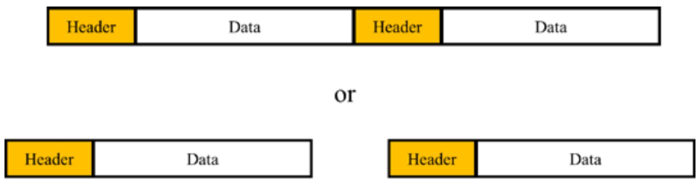

Figure 3 Adding a Header

Frame Detection. When the receiver receives enough samples, the first thing is to determine whether there is something transmitting. One method is to correlate the predefined header with the received samples. When the samples contains a frame, correlation will have high energy. Use this feature as the indicator of the occurrence of a frame.

Synchronization. Once the occurrence of a new frame is confirmed, the next thing is to find out the accurate boundaries of the symbol. You can develop your own way. One suggested way is to leverage correlation again. A unique and long header can help to find out the accurate boundary of the frame. See the Matlab example to see how to design a unique header.

Demodulation. Demodulation is the inverse process of modulation. For PSK, multiply the received signal by the carrier wave will reveal the conveyed symbols. See the lecture slides or reference links for more information.

Tips:
a. Do not use very long frames. This is because the ADC/DAC rates of different devices are slightly different (i.e., frequency offset). For a one-second recording, the number of samples from different devices may have differences up to ~100 samples, which will affect the synchronization for long frames.
b. Do not use a very short header. Speaker takes time to warm up (see ringing effect [1]).

Checkpoints:
The group provides two devices: NODE1 and NODE2
CK1(5 points). The TA provides a TXT file "INPUT.txt" which contains 10000 "0"s or "1"s. NODE1 sends bits according to this file. NODE2 stores the received bits into a TXT file "OUTPUT.txt". During the transmission, TAs keep quiet.

The transmission must be finished within 15 seconds.

<15s	-0%
>15s	-100%

TAs compare the difference between INPUT.txt and OUTPUT.txt through a tool similar to "diff":

<80%	-100%
80% to 99%	-20%
>99%	-0%

(Tasks with "Optional" tag are optional tasks. The instructor is responsible for checking and grading the optional tasks. Contact the instructor to check if you have finished one or more of them. Part 4 is checked by TAs)

Part 4. (Optional +2 points) Error Correction

Current acoustic link is fragile to noise. You can add redundancy in your bits to resist errors caused by noise. This procedure is called coding. There are many coding schemes. Refer to [5] for more information (Convolutional code and Fountain code are suggested ones in this project).

Checkpoints:

The group provides two devices: NODE1 and NODE2

CK1(2 points). The TA provides a TXT file "INPUT.txt" which contains 10000 "0"s or "1"s. NODE1 sends bits according to this file. NODE2 stores the received bits into a TXT file "OUTPUT.txt". During the transmission, TAs will clap their hands for four times.

The transmission must be finished within 30 seconds.

<30s	-0%
>30s	-100%

TAs compare the difference between INPUT.txt and OUTPUT.txt through a tool similar to "diff":

<100%	-100%
100%	-0%

Part 5. (Optional + 4 points) Higher Bandwidth

The Bandwidth of your acoustic link is limited by many factors. A big one is the echoes. Echoes (i.e. signals from multiple paths) blur the boundary between the symbols/bits. Handling echoes is a challenging task. One way is to estimate the property of the echoes and try to cancel it out (see equalization [2]). Sometimes we can leverage advanced modulation (see OFDM [3]) to reduce the impact of the echoes. Some researchers have already given a good example on how to implement OFDM in acoustic channel [4]. I suggest you take a look at it. This article provide useful information for your implementation even if you are not going to implement OFDM.

Checkpoints:
The check routine is similar to Part3.

Part 6. (Optional + 1 point) Need for Speed

Do not use float, double, division, multiplication and complex math functions such as sin() and cos() in your program.

Tips:
a. Use integer and bit shift operations [8].
b. Use look up table to implement complex functions.

Checkpoints:
The check routine is similar to Part3.

Reference and Useful Links

[1] Ringing Effect https://en.wikipedia.org/wiki/Ringing_(signal)
[2] Equalization https://en.wikipedia.org/wiki/Equalization_(communications)
[3] OFDM https://en.wikipedia.org/wiki/Orthogonal_frequency-division_multiplexing
[4] Acoustic NFC http://dl.acm.org/citation.cfm?id=2534169.2486037
[5] FEC https://en.wikipedia.org/wiki/Forward_error_correction
[6] ASIO Audio Interface https://manual.audacityteam.org/man/asio_audio_interface.html
[7] ALSA http://www.alsa-project.org/main/index.php/Main_Page
[8] Bit shifting and adding https://stackoverflow.com/questions/2776211/how-can-i-multiply-and-divide-using-only-bit-shifting-and-adding
[9] Digital Modulation https://en.wikipedia.org/wiki/Modulation#Digital_modulation_methods
[10] ASIO4ALL www.asio4all.org
[11] JASIOhost https://github.com/mhroth/jasiohost
[12] RTL UTILITY https://oblique-audio.com/rtl-utility.php